Pattern Matching Syntax

This is for the pattern matching rules and grammar for structuring patterns. For understanding abstract syntax for defining the grammar used here, go to **WebAssembly Conventions.**

Lexical Format

PatternStr	::=	(Pattern)$^+$	
Pattern	::=	Exact | ManagedCode	
Exact	::=	(c: Echar)$^+$	
Echar	::=	c: char	(if c ≠ '<')
	|	'\\<' ⟹ '\<'	

Note that '\\<' actually contains double slash, as this can cause confusion due to writing '\\' in programming language IDEs for the computer to interpret it as a single slash '\'.

Code ::= Reference | Free | Range | Composition | Choice

When parsing a pattern matching code, the code's type is determined by what of these choices (Reference, Free, …etc) is compatible with the pattern, if more than one satisfy this, the one on the most left in the definition is chosen (so Reference would be chosen over Free for example).

Choice ::= '[' (c_1: char)$^+$ (',' (c_2: char)$^+$)* ']'

This code represents a choice where the text section can be any of these choices. For example, '[a,b,c]' allows the letter to be either 'a', 'b', or 'c'. The choices can be of any length, except when negation is applied to the code, and in that case, all choices must have the same length. The reason behind this is for it to be able to return a length when it matches, which is when the text is not any of the choices (in case of negation), for non-negated code, just return the length of the choice that matched (or the largest choice in case there were multiple matches).

```
Composition  ::=  '[' (c1: ManagedCode) ((o: Op) (c2: ManagedCode))* ']'
Op           ::=  AND | OR | FOLLOW
AND          ::=  '&&'
OR           ::=  '||'
FOLLOW       ::=  '->'
```

This code combines multiple codes using operations, and it matches when all operations and subcodes match. The AND operation matches when both sides match. The OR matches when any side matches. The FOLLOW matches when the left side matches for the text section, and then the right side matches for the next text section. Note that for FOLLOW, its mechanism is similar to two separate codes ($c_1 \rightarrow c_2 \equiv$ '\<' c_1 '>' '\<' c_2 '>').

Note that there is an interesting change that occurs when applying negation to a composition code. When applying negation, it applies this negation to all operations and subcodes EXCEPT ORDER, where applies to its subcodes only and not itself. In case of AND/OR, it applies like logical gate NOT. If a subcode already has negation applied, it will just reverse it.

```
Range  ::=  '[' (c1: char) '-' (c2: char) ']'
```

This code checks if the letter x satisfies the following condition:

$$UFT16(c_1) \leq UFT16(x) \leq UFT16(c_2)$$

Where UTF16 returns the utf8 code of the letter. For example, '[a-z]' accepts any letter with utf8 code between a and z (both inclusive), which is all small case letters.

```
Free  ::=  '[]'
```

This code just indicates that any letter is accepted. Note that in negation, this code normally becomes useless as it means that it accepts nothing, so it was reprogrammed to represent End Of File (EOF). In other words, it will return 'valid pattern' when there is no more text sections to check for patterns. For example '[[a-z](+)->[](~)]' will accept a string consisting of a sequence of small case english letters (at least one character) at the end of the file, thus 'abc' is valid, but 'abc2' is not as after the letters, it checks if it is EOF.

Reference	::=	‘#’(c: idchar)$^{+}$
idchar	::=	‘a’ | … | ‘z’
	|	‘A’ | … | ‘Z’
	|	‘0’ | … | ‘9’

This code converts to reference the ManagedCode object, that already exists or will, that has the same CName (without $) as the substring after ‘#’ in Reference. This is mainly used for recursion, where a composition code has a reference for itself as a subcode.

ManagedCode	::=	‘\<’ (n: CName)$^{?}$ (f: FName)$^{?}$ (c: Code) (s: Setting)$^{?}$ ‘>’
CName	::=	‘$’ (c: idchar)$^{+}$
FName	::=	‘%’ (c: idchar)$^{+}$
Setting	::=	‘(’ (m: Mod) (‘,’ (n: NumRange))$^{?}$ ‘)’ (if m = ‘~’)
	|	‘(’ (m: Mod) ‘)’
	|	‘(’ (n: NumRange) ‘)’
Mod	::=	‘~’ (‘,’ (‘?’ | ‘*’ | ‘+’))$^{?}$
	|	‘?’ | ‘*’ | ‘+’
NumRange	::=	(n_1: num) (‘,’)$^{?}$
	|	(n_1: num) ‘,’ (n_2: num)
num	::=	(‘0’ | … | ‘9’)$^{+}$

The CName represents a unique name for that code instance to be referenced if needed.

FName is the Feature Name of the code instance. The feature system is a way to extract features from the pattern text when validating it. It works by adding the text that the ManagedCode validates (meaning it follows its pattern) to a list in a map where the FName of that code instance is the key, and the list is the value. The list is used for the case when that ManagedCode is repeated multiple times, and in that case, all text that follow the code pattern are added to the list. For example, if ‘\\<%test[0-9](+)>’ is the ManagedCode, and ‘248’ is the input text, then the map will be{ test: [‘2’, ‘4’, ‘8’]}. Notice that every digit is stored individually, this is because the code itself checks for one character. If you want all of them to be in a single text you can wrap the code with Composition: ‘\\<%test[[0-9](+)]>’.